Palo Alto Research Center

# Schemes: A High Level Data Structuring Concept

by James G. Mitchell and Ben Wegbreit

XEROX

# Schemes: A High Level Data Structuring Concept

by James G. Mitchell, Ben Wegbreit

CSL-77-1 January 1977

In recent years, programming languages have provided better constructs for data type definitions and have placed increasing reliance on type machinery for protection, modularization, and abstraction. This paper introduces several new constructs which further these ends. Types may be defined as similar to existing types, *extended* by additional properties. *Schemes* are type-parameterized definitions. For example, symbol tables and symbol table operations can be defined as a scheme with the key and value types as parameters; an instantiation of the scheme implements a specific type of symbol table. Because new types are typically defined along with other related types, an instantiated scheme may *export* a set of new types. A set of schemes with a common name and common external behavior can be viewed as alternative implementations of an *abstraction.* Parameter specifications associated with each scheme are used to select the appropriate implementation for each use.

Key Words and Phrases:

data types, schemes, polymorphism, programming languages, abstractions

CR Categories:

4.12, 4.13, 4.22

XEROX

PALO ALTO RESEARCH CENTER
3333 Coyote Hill Road / Palo Alto / California 94304

## 1. INTRODUCTION

The evolution of programming languages has seen a steady development in the use of data types. Early languages provided only a fixed set of types; later ones introduced type definition facilities; more recently, types have begun to be used for program abstraction and modularization, e.g., in Simula 67 [Dahl]. In this paper, we continue this development. We describe several properties desirable in a type system, which are not accommodated in existing languages. We then describe a new program structure, the *scheme*, which provides them.

Much of the research in data types has been driven by a few sample problems which have provided a focus for programming-language research [Hoare, Dahl]. A popular example is the notion of a list of objects: the characteristics of lists which we wish to capture are that they represent homogeneous, possibly empty sequences of objects which may grow and shrink dynamically with the insertion or deletion of objects. Since list structures generally involve the use of pointers, they are one of the more "dangerous" data structuring mechanisms in a programming language [Hoare]. One of the aims of this paper is to present mechanisms by which the notion of "lists of objects" could be programmed once and for all and protected against misuse. We will call a group of definitions and programs which cover a notion such as this an *implementation scheme*, or simply a *scheme*. The purpose of this work is to discuss how schemes such as "list of objects" can be added to a high-level programming language.

Other examples will involve complex numbers, the notion of a reference-counting storage allocator for any type of objects, and a family of hash table schemes. We will first use the list example to outline a number of the properties which we desire for schemes.

## 2. Properties of schemes

### *2.1 An Example - Singly Linked Lists*

The first question to be answered is: Why is it not already possible to define notions such as "list of objects" in existing languages? Suppose that an assembly-language programmer wanted to write a package of routines which could make and manipulate singly linked lists of any sort of values. He would have to establish a set of parametrized procedures to copy list cells which would either take as a parameter the number of words occupied by the datum in a cell or possibly the address of a user-supplied routine for allocating and copying those values. On this framework he could build the list processing capabilities.

This approach suffers from four drawbacks. Firstly, if a single set of routines is to provide this capability, then many of them will have to be given extra parameters on each call to specify information such as the number of words in the datum; we would like to avoid such run-time overhead. Secondly, because assembly language is untyped, there are no guarantees that programs will supply such size parameters correctly, or will not attempt to make lists containing two different kinds of values: we would like the protection offered by a typed language. Thirdly, although the list processing machinery could all be written in one physical unit, there is absolutely nothing to prevent *any* routine from maintaining its own pointers to list cells, or from smashing the pointers in list cells: we would like to confine access rights for a given data structuring mechanism to textually localized regions. Fourthly, in the absence of types, one cannot have common function names for similar operations on many different types of data; i.e., there can be only one procedure with a

given name in a program. Hence, if one wants a function which yields the length of some object, he must name it differently for each object type: we would like a more generic naming mechanism.

To summarize, the methods of assembly-language programming require four improvements.

(R1) There should be a minimum of run-time overhead incurred by a general list processing package.

(R2) The language should be fully typed to prevent misuse of a general package.

(R3) It must be possible to confine the access to certain parts of a data structure to a specific set of routines which are held responsible for it.

(R4) It should be possible to reuse the same names for similar functions.

A high level language does not necessarily overcome these problems. For instance, typeless languages such as Bcpl [Richards] inherit the same drawbacks as assembly-language. Further, Bcpl imposes a language restriction that a function can return only single-word entities. Thus, any generalized implementation scheme in Bcpl will require passing pointers among procedures instead of actual values. This is perfectly acceptable for the list example; for some of the other examples discussed later, it is not.

PL/I [IBM1] is partially typed in that non-pointer objects have a rigid type structure whereas pointers are effectively typeless. Thus, a technique like that suggested for Bcpl can be used for the concept "list of", at the expense of giving up type protection. PL/I did, however, take a useful step forward in introducing *generic functions*, which allow a single name to be used for a number of operations, with the correct one being chosen on the basis of the types of its arguments.

In many ways, Simula 67 represents the most significant attempt to provide a means of associating a new set of behaviors with previously defined types. Simula *classes* provide several of the properties desired for schemes. A class defines a set of value *components* and operations, together called the *attributes* of the class, a terminology which we will adopt for schemes as well. Any type in the language can be *prefixed* with a class name, meaning that it will have both the attributes of that class and those defined for the new type itself. The language is almost fully typed (there is an omission in the case of procedure variables), and much of the type checking is done at compile-time. However, it lacks several other desirable features explained below:

(R5) One should be able to write *polymorphic* definitions.

(R6) One should be able to add new qualities to a given type in a more general manner than Simula's strict prefixing.

(R7) One should be able to deal with actual objects, not just pointers to them.

Classes in Simula would be truly *polymorphic* if they could have type parameters. Then one could define a class *stack*, for instance, in which the type of the objects to be the values on the stack was a parameter of the class *stack*. As it is, the ability to prefix one class with another does give a limited form of polymorphism. We would, however, like to be able to associate a number of attributes with a type in other ways than Simula's strict subclass capability. For instance, while it is possible in Simula to define a class *list* and then to use it to define lists of objects of any given type, it is not possible to define a class so that it can use the list notion twice and define objects which are simultaneously on two singly threaded lists. We would like to have a more general mechanism for associating groups of attributes with a type.

In Simula, one cannot create local variables of a given class, only pointers to such objects; the objects themselves must be explicitly created and are reclaimed by garbage collection. For many applications this is exactly the *right* thing to have, but it does introduce an asymmetry in the data structuring mechanisms, and it prohibits certain kinds of classes, for instance, simple, multi-component objects such as complex numbers, which could be created as normal local variables.

The incompleteness of Simula's typing system was rectified in Algol 68. Other than this, however, Algol 68 [vanWijngaarden] is less suitable for our purposes than Simula because it has no mechanism like classes for encapsulating all the operations on the objects of some type, nor any means such as prefix classes for extending the behavior of previously defined types. Algol 68 has extended the capability for generic operations to include user-defined operators for new data types, although the programmer cannot define generic procedures.

EL1 [Wegbreit] treates types as manipulable values of the language. Types can be computed as the results of expressions, compared and assigned to type-valued variables. One can write functions which take types as arguments and return types as their values. Such type-generating functions provide some of the facilities of schemes. For example, it is straightforward to write a function which computes "stacks of *t*" for any type-argument *t*. EL1 also provides generic procedures and a partial framework for encapsulating a type and restricting access to it. It lacks a subclass mechanism such as Simula's and its generic procedures are less convenient to write than the generic operators of Algol 68.

Other current language research efforts, [Gries-Gehani], Clu [Liskov-Zilles] and Alphard [Wulf] deal with the notion of parametrized types, encapsulation of types, and extending built-in operators to new types. These languages provide alternative solutions to several of the problems we treat in this paper.

### *2.2 Other Properties of Schemes*

While reviewing the characteristics of several languages above, we outlined some of the properties which we would like schemes to have. There are some other properties which did not arise in that discussion, but which we deem important:

(R8) A scheme should be able to define and export a set of related types, rather than a single type. A single type is not always sufficient to describe objects with complex behaviors.

(R9) Schemes should cascade nicely: for instance, given a scheme which can append the behavior of a reference-counted allocation scheme to any type, and a scheme to implement lists of objects, one would like to be able to define a new scheme which provides lists of reference-counted objects simply by using the already existing ones without explicitly having to program their union.

(R10) It must be possible to impose constraints on the parameters to a scheme. In order to select one scheme from a set of schemes for symbol tables, for example, some selection on the basis of the parameters to a generic scheme is necessary.

(R11) It should be possible to completely specify the behavior of any defined type. In particular, it should be possible to specify how objects are to be initialized, how assignment and other operations are to be interpreted, and how objects are finalized and destroyed. A scheme definition should specify this behavior for a family of types.

### *2.3 An Outline of the Scheme Mechanism*

Before going into detail, it will be useful to sketch the main points of the mechanism we propose. As a starting point, we take a fully typed language which has a protection mechanism to localize access authority for a type to its defining form [Morris]; for our purposes, this will be Pascal-based [Wirth2]. A *scheme* is a new language form which can be parameterized. The parameters to a scheme may be types, procedures, and other values which are known at compile time. (Such values are sometimes called "manifest"). To a rough approximation, a scheme may be regarded as a macro. Each use of a scheme is expanded during compilation to produce an *instance* of the scheme. A scheme instance is typically a type, e.g., "list of integers", or "reference counted strings", but it can be any compile-time constant, such as a record type whose components are types, so that a single scheme can define a set of types. Schemes entail no run-time overhead because they are instantiated during compilation. Since they are parameterized, a single definition can be used to specify the behavior of a family of types. Finally, it is possible for multiple schemes to have the same name (if, for example, they represent different implementations of the same abstraction). A generic selection mechanism will be used to choose the correct scheme, based on properties of the instantiating arguments.

The remainder of this paper is devoted to a design for schemes as an addition to the Pascal programming language [Wirth2]. We have undertaken this as an extended foray into uncharted waters. We have made little effort to be conservative; indeed, we have introduced new notation and language features whenever they seemed useful. Our aim here was not to consolidate the known, but to explore the unknown. Those points where the exposition is labored often mark shoals where subsequent efforts might well be spent. Further, this is at present a paper design: no implementation has been undertaken. Our previous experience with designs for programming language features leads us to think that errors will be discovered and changes will be made in an implementation, but also that the main ideas will prove useful. Therefore, it seems appropriate to present the design in its present state.

## 3. Some extensions to Pascal

When carrying out this design, we discovered the need for several extensions to the Pascal type system. These extensions were required to support the scheme system, but could also be used in defining individual types. They serve to facilitate definitions which would otherwise be awkward or lengthy. Because they can be presented in the context of a conventional type system and build on existing techniques [Geschke-Mitchell, Wegbreit] in a straightforward way, it will be simplest to start with them.

Suppose we wish to define the type *Rational* whose components are two integers *numerator* and *denominator*. A new *Rational* is to be initialized to zero, represented by the pair (0,1). A procedure is to be provided for allocating *Rationals* from a free storage pool and the usual arithmetic operations are to be implemented. The definition follows. The reader is advised to skip immediately to the explanatory comments following it:

```
type Rational = record                                              (m1)
    {Rational maintains the invariant, denominator > 0}
    private var numerator, denominator: integer;                    (m2)
    procedure INIT(shared p: Rational);                             (m3)
        begin
        p.numerator := 0; p.denominator := 1;                       (m4)
        end;                                                        (m5)
    function NEW returns p: Rational.ref;                           (m6)
        begin
        p.lower := new Rational.lower;                              (m7)
        INIT(p);                                                    (m8)
        end;
    function PLUS(x,y: Rational) returns z: Rational;               (m9)
        begin . . . end;
    function Positive(x:Rational) returns Boolean;                  (m10)
        begin
        return (x: numerator > 0);                                  (m11)
        end
    . . .
    end Rational;                                                   (m12)
```

The line numbers in the right margin are not part of the program; they serve as keys to the explanations given in the following sections.

### *3.1 Extensions to Pascal records*

We have made four changes to Pascal records. One is trivial and was added solely to enhance readability: the **end** for a record type may be followed by an identifier as in (m12) to say which record definition it acts as right bracket for. The second change is more consequential: the field list in a record may contain any declarations, including procedures (m3), functions (m6), constants and types, much like modules in Modula [Wirth3] or Euclid [Euclid1]. The third change allows some components of a record type to be declared **private** (m2), meaning that they are not accessible from outside the record definition. The fourth generalizes the notation for referring to components of records: if a record type *R* defines a constant component, *c*, then the notation *R.c* may be used to refer to it, since it has the same meaning in all objects of that type.

### *3.2 Changes to Pascal procedures and functions*

In Pascal, parameters to procedures or functions may be passed in one of two basic ways: by value and by reference (**var**). We have changed this to the following three *parameter-passing modes*:

**value**: the actual parameter can be any expression (of which a simple variable is a special case); its value is copied into the parameter list for passing to the procedure or function.

**const**: the parameter is passed by value, just as above, but its value may not be altered in the body of the procedure or function for which it is a formal parameter.

**shared**: the parameter is to be passed by reference; changes to the formal parameter will change the actual parameter. The actual parameter must specify a variable and may not be a general expression.

The default parameter-passing mode is **value**. These modes are orthogonal to the types of the formal parameters.

We have expanded the possible value ranges of Pascal functions: here, functions may be declared to return a value of any fixed type. Also, we have altered the syntax for specifying the type of the return value to have one of the two forms:

**returns** <identifier> : <type identifier>
**returns** <type identifier>

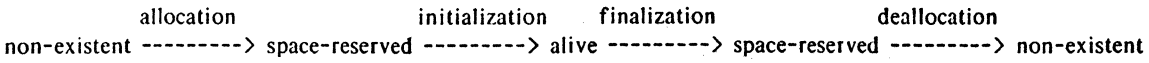
and have added a return statement of the form

**return** <expression>

which may appear anywhere in a function. If the result of a function is given a name (by using the first form for the **returns** specification), then a **return** statement is not necessary, and the last value assigned to that variable will be the value returned from the function. This syntax is copied directly from Euclid [Euclid1].

### *3.3 The lifetimes of variables and values*

Pervading this report is a view of the lifetime of a data object, whether associated with a variable or not. This lifetime divides into three states: *non-existent*, *space-reserved*, and *alive*. The process by which an object comes into existence, is used, and then is destroyed uses the transition operations *allocation*, *initialization*, *finalization*, and *deallocation* as shown in the *lifetime diagram* below:




A properly typed object is only accessible outside its definition when it is in the *alive* state. The four transition operations are characterized by the following:

*Allocation*: Space is acquired for the object. This space can come from two different kinds of storage: (1) the object can be embedded in an activation record, a record, or an array; or (2) it can be allocated separately from some free storage pool. If the former, the object's lifetime is intimately tied to the surrounding record or array, and we call it an *embedded* object. If the latter, its lifetime is strictly under program control, and we call it a *free-standing* object.

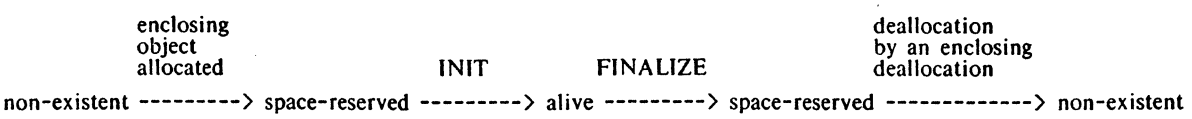
*Initialization*: The newly acquired space is assigned initial value(s) in order to establish some invariant property of the object. All built-in data types have an associated, system-defined initialization. All pointers are initialized to **nil**, but the initial values for integers, sets, or subrange objects are undefined.

*Finalization*: At the end of its useful lifetime, any housekeeping is done for the object before its space is freed for reuse. For example, the object may possess references to global objects which are to be explicitly destroyed before it can be considered finalized; if so, they would have to be finalized and deallocated as part of its finalization.

*Deallocation*: The object's space is freed. For an *embedded* object, this is a null operation because its storage goes away as part of deallocating the enclosing object. For a *free-standing* object, this will require a specific action to return the space to some storage pool.

This outline for each object's lifetime is reflected into our extended Pascal as follows. Any record type may contain procedures named NEW, INIT, FINALIZE, and FREE (with suitable parameters, described below). In general, the user may supply none, some, or all four of these procedures: if he omits one, the system will supply a default meaning for it.

An *embedded* object is created when its enclosing object is. In this case, NEW and FREE of the embedded object's type are not used. However, if INIT is present, it will be invoked automatically after allocating the record or array in which the object is embedded. Before the surrounding array or record is freed, any FINALIZE procedure for that object will also be invoked automatically. This is shown in the following annotated *lifetime diagram*.




*Free-standing* objects are created when the program uses the **new** operator, which takes the form:

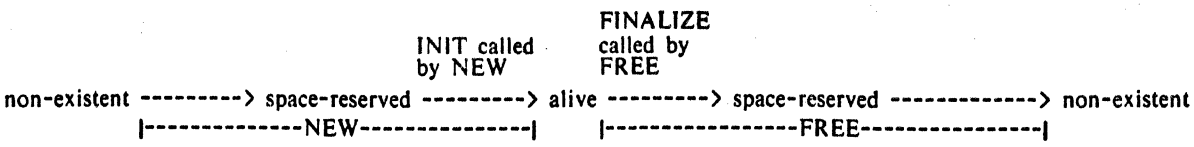
**new** <type expression> <parameter list>

This compiles into a call on the NEW function for the specified type with the additional parameters used to initialize the object. The value to be returned by the NEW function must be a reference type, pointing to the newly allocated and initialized object. In this case, the NEW function is itself responsible for initializing the new object (*otherwise, it couldn't return a pointer to a valid object*). That is, the NEW function, if provided by the programmer, is to perform the combined operations of allocation and initialization, the latter by calling INIT. The system-supplied default NEW for objects of a given type also initializes them; further, programmer-supplied initialization can be combined with system-supplied allocation by writing only an INIT procedure: if a given type specifies INIT but not NEW, the system will be responsible for allocating space for it, but will call the user-supplied INIT to correctly initialize it before returning a reference value as the result of the **new** operation.

A free-standing object for which the program has a reference can be freed by using the **free** operation which has the form:

**free** <reference value>

This is the dual of **new**, and will compile into a call on the type's FREE procedure, if present. In parallel with the New function, FREE is responsible for finalizing the object (by calling FINALIZE) before deallocating its space. If no user-specified FREE is given for a type, then the system will manufacture a call on that type's FINALIZE (if present) before deallocating the object. The *lifetime diagram* is




The routines NEW, INIT, FINALIZE, and FREE are called the *lifetime routines.* They are generic and must have the following forms for each type *T*:

**function** NEW <parameter list> **returns** *T*.**ref** {returns a pointer to a *T*}
**procedure** INIT (**shared** *T*)
**procedure** FINALIZE (**shared** *T*) {Note the parallelism with INIT.}
**procedure** FREE (*T*.**ref**) {A free-storage package needs the address of the object to be freed.}

In the case of NEW, these specifications do not constrain the types or number of its parameters.

In the *Rational* example, both an INIT procedure and a NEW function are provided (m3 and m5), and the NEW function explicitly calls INIT (m8). The type of the result returned by NEW is specified as *Rational*.**ref**: for every type, *T*, there is an associated type *T*.**ref**, which is the type for references to objects of type *T*. We denote it as if **ref** were an attribute (a **type** attribute to be sure, but an attribute nevertheless). It is occasionally useful to explicitly define **ref** for a record type, replacing the system-supplied attribute; in such a case, *T*.**ref** has the explicitly specified meaning. This is a way to define together a type and a special class of pointers to objects of that type. The type name then serves as a handle on both.

Conversely, for a given reference type *R*, there is an associated type, *R*.**node** which is the type of the objects to which an *R* refers. Consequently, for any type *T*, *T*.**ref**.**node** = *T*.

In the example, NEW obtains storage for a *Rational* by making a **new** *Rational*.**lower**. For each record type *R* which specifies any non-default behavior, there is an associated record type *R*.**lower** which has the same structure, *but entirely standard, system-defined behavior* (i.e., no user-supplied NEW, INIT, FINALIZE, or FREE). We call this associated type the *foundation type* of *R*. To ensure type integrity the attribute **lower** cannot be applied outside the scope of *R*. The value returned from the operation **new** *Rational*.**lower** (m7) is a *Rational*.**lower.ref**, which is basically just a reference to a record of two integer components. It is assigned to *p*, which is treated as if it had type *Rational*.**lower.ref**; that *p* is to be so treated is indicated by writing *p*.**lower**.

### *3.4 Records, procedures, and generic selection*

There are three ways in which record attributes may be invoked:

(1) with the dot notation, e.g., *Rational*.**ref**,

(2) with the generic selection mechanism when triggered by an appearance of a procedure name, e.g., *Positive*,

(3) with the generic selection mechanism triggered by the appearance of a language-established operation, e.g., **new** invokes NEW, ":=" invokes ASSIGN, "+" invokes PLUS, etc.

(4) with the generic selection mechanism when triggered by a system action, e.g., the procedure INIT associated with object initialization.

The generic selection mechanism is invoked by the compiler when it encounters a procedure name *p* applied to arguments whose types are *t1, . . ., tn* in a context requiring result type *tr.* It looks for an interpretation for *p* whose type structure agrees with *t1, . . ., tn, tr.* If a unique such meaning is found, it is used; if no meaning is found or multiple meanings are found, the expression is flagged as as error [Geschke-Mitchell]. For example, the function PLUS (m9) in *Rational* takes two *Rationals* and returns a *Rational* value; it is thus suitable for use in any expression of the form "*exp1+exp2*" in which *exp1* and *exp2* are both *Rational* expressions. Assignment is treated specially in one regard: If the types of the left and right sides are *L* and *R*, respectively, then an interpretation is sought in which the parameter type structure is (*L*; *R*) and *L*'s parameter-passing mode is **shared.**

Note that PLUS is allowed to access the private components of a *Rational* because it is declared within the type. This, along with generic selection, makes it possible to write binary operators dealing with the internal representation of separate values of the same type. This convention might be contrasted with that used in extensions to Simula 67 and various encapsulation proposals based on Simula classes [Palme][Liskov-Zilles]. In these proposals, *x.f(y)* can be used to invoke the *f* procedure defined in *x*'s class. *f* has normal access to the parameter *y* and privileged access to *x* (because *f* is in *x*'s class). Privileged access to *y* is not allowed.

We have repeatedly used the idea that each record type carries with it a set of attributes, some defined explicitly, some provided by the system as part of every record type, and some explicitly supplanting a system-provided definition. This structure serves two ends: that of providing a reasonable behavior for any defined type with minimal specification in the definition, and that of allowing additional specifications to override the system defaults. One special case of overriding deserves comment: a definition which renders illegal the use of a system-defined attribute. This may be accomplished by defining an attribute with the value **Error**; use of that attribute will be detected as illegal during compilation. For example, a system-supplied attribute such as **ref** may be overriden by a declaration such as

```
type ref = Error;
```

*3.5 Extending the semantics of an object*

As a matter of notational convenience, it is often useful to define a new type by extending an existing one. This may be accomplished by the construction

```
type Trec = basePart: BaseType extended by <field list> end Trec
```

or

```
type Trec = BaseType extended by <field list> end Trec
```

where <field list> is the syntax which normally appears in the context **record** <field list> **end**. This is defined to be shorthand for

```
type Trec = record
    basePart: BaseType {or some unnamed BaseType component}
    <field list>
    procedure ASSIGN (shared x: BaseType; y: Trec)
        begin x := y.basePart end;
    end Trec
```

*Trec* is thus defined to have a preferred *BaseType* component, additional attributes as specified, and a conversion from *Trecs* to *BaseTypes.* In this sense, *Trecs* behave as proper

extensions of *BaseTypes.* The user program may supply its own ASSIGN routine in the <field list> to override the one supplied automatically as part of the **extended by** construction.

For example, consider extending *Rational* so that each rational keeps a count of the number of times its value is changed by assignment.

```
type TallyedRational = rat: Rational extended by
    var tally: integer;                                              (t1)
    procedure INIT(shared p: TallyedRational);                       (t2)
        begin
        p.tally := 0;
        end;
    procedure ASSIGN(shared left: TallyedRational; right: Rational);  (t3)
        begin
        left.tally := left.tally+1;
        l.rat := right
        end;
    end TallyedRational
```

This defines *TallyedRational* to be like *Rational* except in three ways:

(t1) All instances have an additional component named *tally,* of type **integer.**

(t2) An INIT is supplied for *TallyedRational.*

(t3) A procedure named ASSIGN is supplied to be used when the left hand side of an asssignment is a *TallyedRational.*

# 4. Example schemes

## *4.1 An Implementation of Queues*

As a first example we define a scheme which implements queues using a linked list of nodes which hold queue elements. Each queue has a head which contains pointers to the front and rear nodes. To make the treatment of empty and non-empty queues uniform, we adopt the convention that all queues end with a dummy node. Thus, the empty queue is represented by a queue head in which the front and rear pointers point to a dummy node. When a queue head is initialized, it is set to this empty state. Adding an element to a queue is accomplished by putting the element into the dummy node, allocating a new node to serve as dummy, putting it on the end of the node list, and adjusting the *rear* pointer in the queue head. Removing an element from a queue is accomplished by setting the *front* pointer in the queue head to reference the second element in the node list and returning the element found in the former first element.

The following scheme defines a record having two **type** attributes: *Head* and *Node*; also, it includes an initialization routine and the set of queue operations.

```
scheme Queue(type ItemType) = record
    type Head = record var front, rear: Node.ref end;
    type Node = item: ItemType extended by private var next: Node.ref end;

    procedure INIT(shared q: Head);
    {This will be invoked whenever a new Head is created.}
        begin
        q.front := new Node;
        q.rear := q.front
        end;

    function Empty(q: Head) returns Boolean;  {Test for the empty queue.}
        begin return (q.front = q.rear) end;

    procedure Add(shared q: Head; item: ItemType);
    {Add item to q, modifying q as a side effect.}
        begin
        q.rear↑.item := item;
        q.rear↑.next := new Node;
        q.rear := q.rear↑.next
        end;

    function Remove(shared q: Head) returns item: ItemType;
    {Remove (and return) the first item from q, modifying q as a side effect.}
        begin
        if Empty(q) then Error;
        item := q.front↑.item;
        q.front := q.front↑.next;
        end;

    end Queue;
```

To use this scheme in defining a type, we instantiate it; for example

```
type IQ = Queue(integer);
```

defines *IQ* to be an instantiation of *Queue*. From the definition of *Queue*, it follows that *IQ* is a **record** type having attributes *Head* and *Node*, which are both types. Thus, *IQ.Head* is the data type for heads of queues which hold integers, and

*q1: IQ.Head*;

declares *q1* to be a variable of this type. It would, of course, be possible to contract this to the equivalent

*q1*: *Queue*(**integer**).*Head*;

Each instance of the *Queue* scheme is a separate data type which has, at least in principle, its own copy of the scheme procedures INIT, *Empty*, *Add*, and *Delete*. We consider INIT as representative. In the instance *Queue*(**integer**), the INIT procedure is defined so that *Head* in its formal parameter specification is instantiated to *Queue*(**integer**).*Head*. Thus the formal parameter *q* in INIT is passed as a **shared** parameter and has the data type

```
record
    var front, rear: Queue(integer).Node.ref
end
```

When *q1* is created, the INIT procedure will be invoked by the system. It will be seen that the types are correct. The code in INIT allocates the dummy node and links it in. If we then execute

*Add*(*q1*, 9)

the instance of *Add* in the type *Queue*(**integer**) is invoked. Similar considerations apply to the other *Queue* operations.

It is important to note that the *Queue* implementation scheme is not itself a type, but rather it serves as a template for a set of (related) data types, namely all its instantiations. There are, of course, other ways of implementing queues; they may be described by other implementation schemes.

### *4.2 Reference-counted Objects*

Next, we define a scheme for controlling storage allocation of objects using a reference count added to each object. The reference count keeps track of how many *pointers* refer to the object at a given time. When another pointer is made to refer to the object, its count is incremented, and when a pointer no longer refers to it, the object's count is decremented. If an object's count ever becomes zero, it is deallocated.

```
scheme ReferenceCounted(type ItemType) = item: ItemType extended by    (r1)
    var count: integer;                                                (r2)
    procedure INIT(shared p: ref);                                     (r3)
        begin p.lower := nil end;
    procedure INIT(shared rcItem: ReferenceCounted);                   (r3a)
        Error; {Creating an embedded, reference-counted object is illegal.}
    function NEW returns(newp: ref);                                   (r4)
        begin
        newp.lower := new ReferenceCounted.lower;
        newp↑.count := 1;
        end;
```

```
procedure FINALIZE(shared p: ref );                                    (r5)
    begin
    if p = nil then return
    p↑.count := p↑.count-1;
    if p↑.count = 0 then free p;                                       (r6)
    end;
procedure ASSIGN(shared left, right: ref);                             (r7)
    begin
    if right ≠ nil then right↑.count := right↑.count+1;
    FINALIZE(left); {Finalize left side before it is smashed.}         (r8)
    left.lower := right.lower; {Smash left.}
    end;
end of ReferenceCounted;
```

Line (r1) defines *ReferenceCounted* as a **scheme** which, given a type *ItemType*, defines a new type which extends *ItemType* with some additional attributes. The additional attributes comprise a data component, *count* (r2), and five procedures: two INIT procedures - one for references to reference counted objects, and one for the objects themselves (see explanation below) - NEW, FINALIZE, and ASSIGN, all part of a single **record** added to a *ItemType*. Observe that *ReferenceCounted* is used as a type in line (r3a) within a scheme; the appearance of the scheme's name denotes the current instantiation. Similarly, **ref** denotes *ReferenceCounted*.**ref**.

We will show how the *ReferenceCounted* scheme is used and works by explicating the following code:

```
type RCreal = ReferenceCounted(real);                                  (e1)
r1, r2: RCreal.ref;                                                    (e2)
r1 := new RCreal;                                                      (e3)
r2 := r1;                                                              (e4)
```

Line (e1) defines a new type, *RCreal*, for reference-counted reals; this new type is the result of expanding the *ReferenceCounted* scheme with the parameter **real** supplied. In line (e2), two pointers to *RCreal*s, *r1* and *r2* are defined; their type is equivalent to *ReferenceCounted*(**real**).**ref**.

We will follow *r1* and *r2* through their lifetimes in the execution of the program segment above. *r1* and *r2* are embedded objects, allocated as part of an enclosing frame for the current procedure activation (see section 3.3). The agent who allocates that frame provides space for them. Initialization is done explicitly - by the INIT procedure beginning at line (r3), whose parameter type indicates that it initializes **refs**. We wrote this INIT to demonstrate how a scheme can control *references to objects* as well as the objects themselves. It is in fact redundant because the default initialization for **refs** sets them to **nil**.

In line (e3), a specific request is made to create a new *RCreal* object and to store a pointer to it in *r1*. Since the *ReferenceCounted* scheme has provided a suitable NEW function (r4), it will be called instead of the system-supplied NEW. NEW first invokes the NEW function for the *foundation type RCreal*.**lower** (section 3.3) underlying an *RCreal*. The result of that NEW will be a *RCreal*.**lower.ref** which will then be assigned to *newp*. The appearance of **lower** on the left-hand side causes the system ASSIGN procedure for **ref.lower** to be used. Next, the pointer *newp* is used to initialize the *count* field attached to an *ItemType* as a result of using the *ReferenceCounted* scheme. Then the pointer value is returned and will be assigned to *r1*.

Since *r1* and the value returned by *RCreal*.NEW both have type *RCreal*.**ref**, the ASSIGN procedure is invoked to do the assignment in line (e3). The actual parameters passed to it are *r1* and the newly created *RCreal*, both passed **shared** (i.e., by reference). Since the pointer returned by NEW is not **nil**, the count of the newly created *RCreal* will first be incremented to 2. The FINALIZE procedure is called with the value in *r1* as parameter (line r8). Because the value of *left* is **nil**, no significant actions are performed. Next, the pointer to the new *RCreal* is stored into *r1*, and ASSIGN returns. As the last step of line (e3), the unnamed temporary holding the result of **new** *RCreal* is finalized. The FINALIZE procedure with parameter structure **shared ref** (line r5) is invoked: it decrements the *count* leaving it at 1.

Line (e4) is similar to (e3), and concludes with the object's reference count having the value 2, since both *r1* and *r2* both point to it.

### *4.3 Linked Lists*

Our next example defines linked lists. We provide for simple uses, e.g., singly linked homogenous lists, as well as more complex ones, e.g., nodes with several linkage threads running through them. To obtain this generality, we define the notion of list link with some care and corresponding parameterization. A list link is a pointer to a node. Each node contains a list link as well as other information. The list link may appear anywhere in a node (for there may be many different list links), so we parameterize the procedures for accessing and changing it. Each node also contains an element which is "preferred" in the sense that the node may be considered to be a means for putting together a collection of elements; conversion from the node type to the element type is automatic in appropriate contexts (e.g., when used in a selection operation with "." or "[]"). Let *ItemType* be the type of the preferred element; it is a parameter to the scheme. Given a pointer *p* to a node, it should be possible to explicitly access and change that element, by a qualified reference *p.item*, where *item* is the name given to the element.

In defining the *ListLink* scheme, we must observe these and one other consideration: The actual represention for list links is of no concern to the scheme and we may well wish to use a non-standard one; hence, the type of list link is a scheme parameter, *PtrType*. We require that *PtrType* be defined so that it has four attributes: two procedures, both named *link*, for accessing and changing the link list component in a node; and two procedures, both named *item*, for accessing and changing the non-list part of the component. The *ListLink* scheme is written so that it works properly for *PtrTypes* having these properties, but not necessarily otherwise. It is desirable to state this constraint explicitly. To do so, we use the formal parameter construction

**type** *T* **needs attributes** (<field list>)

The **needs** construction is used to constrain the admissible values for a scheme parameter; an actual parameter which does not satisfy the specification is an illegal use of the scheme. The **needs** clause specifies that *T*, in order to be a valid argument for the scheme, must have the attributes named in the field list. *T* may have other attributes as well, but it *needs* to have at least these.

The following definition establishes that a *ListLink* is a *PtrType* extended by a set of procedures for:

(1) initialization
(2) computing the length of a list
(3) adding a new element onto the front of a list,
(4) appending one list to another,

(5) applying a procedure to each element of a list,
(6) building up a new list by applying a procedure to each element of a list constructing a list of the results,
(7) constructing a new list by copying a list excluding all elements equal to a given item.

```
scheme ListLink
    (
    type ItemType;
    type PtrType needs attributes
        (
        function link(ListLink) returns ListLink;
        procedure link(ListLink; ListLink);
        function item(ListLink) returns ItemType;
        procedure item(ListLink; ItemType)
        )
    )
    PtrType extended by
        function length(value p: ListLink) returns n: integer =
            begin
            n := 0;
            while p ≠ nil do begin n := n+1; p := p.link end;
            end;
        function Cons(newitem: ItemType; y: ListLink) returns p: ListLink;
            {Add newitem to the front of the list y.}
            begin
            p := new ListLink.node;                                    (c1)
            p.item := newitem;                                         (c2)
            p.link := y                                                (c3)
            end;
        function Append(x,y: ListLink) returns ListLink;
            begin
            if x=nil then return y
            else return Cons(x.item, Append(x.link, y))
            end;
        procedure Map(value x: ListLink; procedure F(ItemType));
            {Apply F to each element of the list x.}
            begin
            while x ≠ nil do begin F(x.item); x := x.link end;
            end;
        function MapItem(x: ListLink; function F(ItemType) returns ItemType)
                returns ListLink;
            {Construct a new list consisting of the results of applying F to each
                element of the list x.}
            begin
            if x=nil then return nil
            else return Cons(F(x.item), MapItem(x.link, F))
            end;
        function Delete(item: ItemType; x: ListLink) returns ListLink;
            begin
            if x=nil then return nil;
            if x.item=item then return Delete(item, x.link);
            return Cons(x.item, Delete(item, x.link));
            end;
        . . .
        end ListLink;
```

This is only part of a complete definition. To emulate all the useful properties of Lisp lists, a definition would include procedures for reversing a list, for substituting one element in place of another, finding the n-th element of a list, etc. As the above procedures illustrate the main points, we omit the others in the interest of brevity.

Three examples showing how this scheme is used will serve to explain the definition.

We first consider a simple case. *ListLink* is an implementation scheme for the linking mechanisms for singly linked lists. To use it, we need associated node and reference types. Our first example defines nodes containing integers:

```
type IntNode = record                                   (i1)
    var item: integer;                                  (i2)
    var link: ref;                                      (i3)
    type ref = ListLink (integer, ref.lower)            (i4)
    end                                                 (i5)
```

*IntNode* has two data components, *item* and *link*, with types **integer** and **ref**, respectively. It also has a **type** attribute **ref** that overrides the system-supplied meaning; this **ref** is the result of instantiating the scheme *ListLink*.

The second parameter to *ListLink* is the *foundation type* (section 3.3) for *IntNode*.**ref**. It must possess certain attributes which satisfy its **needs** clause. In particular, **ref.lower** must have associated with it four routines, two of which have the structure

```
function item (ListLink) returns integer            (Ritem)
procedure item (ListLink; integer)                  (Litem)
```

We adopt the uniform referent view in [Ross] and [Geschke-Mitchell] that every data component *d:TypeOfd* of a record type *T* implicitly defines two accessing routines

```
function d(T.ref) returns TypeOfd                   (Rd)
procedure d(T.ref, TypeOfd)                         (Ld)
```

(*Rd*) is used in right-hand contexts to access the value stored in *d*, and (*Ld*) is used in left-hand contexts to assign values to the *d* component. The form of (*Ld*) is derived in [Geschke-Mitchell] because *item* (*x,y*) is taken as an interpretation of *x.item* := *y*. In the case of *IntNode*, its component *item* is treated as implicitly defining the routines

```
function item (ref) returns integer                 (Rd')
procedure item (ref; integer)                       (Ld')
```

as attributes of *IntNode* and as attributes of **ref**. The **needs** specification of **ref**, (*Rit*em) and (*Litem*) are met by (*Rd'*) and (*Ld'*), since line (i4) establishes that **ref** is equivalent to this instance of *ListLink*. Similar considerations apply to (*Ritem*) and the two constraints on *link*.

Our second example using *ListLink* generalizes this, making a scheme from *ListLink* instead of a type. The following scheme implements a general mechanism for singly linked lists of any type of items:

```
scheme SingleListNode(ItemType: type) = item: ItemType extended by
    var link: ref;
    type ref = ListLink(ItemType, ref.lower)
    end SingleListNode;
```

A *SingleListNode* is a record consisting of an *item* of type *ItemType* and a *link* which points to the next *SingleListNode* in the list. The type of *link* is **ref**, which is defined to be the value of the *ListLink* scheme instantiated as shown. The instantiation is similar to that in *IntNode*, above. By defining the attribute **ref** in *SingleListNode* to be equal to the instantiation of *ListLink*, we override the system-defined meaning for **ref** with the procedures defined in *ListLink*.

An individual type such as *ListOfIntegers* can be defined and used:

**type** *ListOfIntegers* = *SingleListNode*(**integer**).**ref**;
*ip1,ip2: ListOfIntegers*;

*ListOfIntegers* is defined to be equal to the **ref** attribute of *SingleListNode*(**integer**). Thus,

*Cons*(13, *ip2*)

invokes the *Cons* routine in *ListLink* with *newitem*=13 and *y*=*ip2*. The expression

*Map*(*ip1, Print*)

applies the procedure *Print* to each integer in the list *ip1*. If *Add1* is a function which takes an integer *x* and returns *x*+1, then

*ip1* := *MapItem*(*ip2, Add1*)

assigns to *ip1* a list in which each element is 1 greater than the corresponding element of *ip2*. Finally,

*ip1* := *Delete*(13, *ip2*)

assigns to *ip1* a list which is a copy of *ip2* with all unlucky elements deleted.

Our third example using the *ListLink* scheme defines a node type which contains two list threads. Doubly threaded nodes can be used for expression nodes in an optimizing compiler [Wulf]. Each expression node is on two lists: the list of syntactically identical expressions and the list of expressions known to have the same value. It should be possible to start from such a node and follow one list thread to visit its set of nodes, or to follow the other list thread and visit its set of nodes. Either thread may be regarded as defining a list, to which list operations such as *Cons*, *Append*, *Map*, *MapItem*, or *Delete* may be applied. There are three classes of pointers to nodes: *LinkType1* which defines one thread, *LinkType2* which defines the other thread, or a simple **ref**. In the latter case, the list operations are *not defined* for the pointer and any attempt to use them is a type error. If *x* is a pointer to a node, the type of *x* determines the interpretation of expressions such as *Map*(*x, F*).

The *DoubleListNode* scheme defines a record type consisting of a component, *item*, of type *ItemType* and two fields to hold the list threads, *link1* and *link2*, having types *LinkType1* and *LinkType2*, respectively. To define the types *LinkType1* and *LinkType2*, the *ListLink* scheme is instantiated twice, with different arguments. The instantiations differ only in the meaning they ascribe to the attribute *link* of the scheme parameter *PtrType*. In the instantiation for *LinkType1*, *link* is tied to the component *link1*; for *LinkType2*, it is tied to *link2*.

**scheme** *DoubleListNode*(**type** *ItemType*) = *item: ItemType* **extended by**
  **var** *link1: LinkType1*;
  **var** *link2: LinkType2*;

```
type LinkType1 =
    ListLink(ItemType, DoubleListNode.ref extended by                    (n1)
        function link(x:LinkType1) returns LinkType1;                    (n2)
            begin return x.link1 end;
        procedure link(x:LinkType1; q: LinkType1);                       (n3)
            begin x.link1 := q end;
        end);
type LinkType2 =
    ListLink(ItemType, DoubleListNode.ref extended by
        function link(x:LinkType2) returns LinkType2;
            begin return x.link2 end;
        procedure link(x:LinkType2; q: LinkType2);
            begin x.link2 := q end;
        end)
end DoubleListNode
```

An instance of this scheme will aid our explanation:

```
type DLInt = DoubleListNode(integer)
```

The node type *DLInt* has three pointer types associated with it: *DLInt.LinkType1*, *DLInt.LinkType2*, and *DLInt*.ref - the first two are explicitly defined attributes while the last is system-defined and not overridden. *LinkType1* and *LinkType2* are two distinct instantiations of *ListLink*, each defining a separate sort of list to which all the list operations are applicable. Consider:

```
dp1: DLInt.LinkType1;                                                    (p1)
dp2: DLInt.LinkType2;                                                    (p2)
dpr: DLInt.ref;                                                          (p3)
dp1 := Cons(11, dp1);                                                    (p4)
```

Because the type of *dp1* is *DLInt.LinkType1*, line (p4) is interpreted as follows: the *Cons* selected is the one defined in the instantiation of *ListLink* specified by the lines between (n1) and (n3). Thus, a new *DLInt* node is allocated (line (c1) of the *ListLink* scheme), the integer component is set to 11 (line (c2)), and the field *link1* is set to *dp1* (line (c3)). That *link1* is set, rather than *link2*, occurs because the *link* procedure of the *ListLink* scheme is bound to the procedure of line (n3). The result of the *Cons* is assigned as the new value of *dp1* (its old value was **nil**). Similarly,

```
Map(dp2, Print)
```

will Print all the integers in the list defined by the second thread.

*MapItem*(*dpr*, *F*) constitutes a type error - *dpr* is a pointer to a *DLInt*, but is not a *ListLink* so list operations cannot be applied to it. The difficulty here is inherent in the expression - there is no way to tell which thread is to be followed by *MapItem*. Also,

```
dp2 := Cons(11, dp1)
```

is not type correct, since *Cons* in this expression returns a value of type *LinkType1* which is not *dp2*'s type.

Suppose we wished to allow expressions such as the latter, with the obvious interpretation. More generally, suppose we wish to provide a mapping from a *LinkType1*.ref to a *LinkType2* with the same referent. This may be accomplished by using a more complex definition of *DLInt* which has a procedure attribute *AsLink2* to perform the conversion:

```
type DLInt = DoubleListNode(integer) extended by
    function AsLink2(x: LinkType1) returns LinkType2;
        begin return x end;
    end;
```

Here we exploit a standard property of protected classes: within a type definition, conversion from a type to its representation and conversely are legal, while outside the defining form conversion is not allowed. With this definition,

*dp2* := *AsLink2*(*Cons*(11, *dp1*))

is type correct, for the function *AsLink2* invokes the conversion procedure. Similary, both *AsLink2*(*MapItem*(*dp1*, *F*)) and *MapItem*(*AsLink2*(*dp1*), *F*) are type correct, with quite different meanings. The conversion routine is, of course, bookkeeping for the type machinery - an optimizing compiler will elide it.

Note that this definition applies only to *DLInt*. Had we instead wished to provide conversion for pointers to any sort of double list node, we would have included the attribute *AsLink2* in the *DoubleListNode* scheme so that all instances of the scheme would have had it available.

### *4.4 Two Way Lists*

One particular sort of doubly-threaded node sees frequent use: the "doubly-linked" or "two way" lists. In such a list, each node points to its predecessor and successor along a single chain. The two threads traverse the chain in opposite directions. If *p* points to such a node then two invariants hold:

**if** *p*↑.*link1* ≠ **nil then** *p*↑.*link1*↑.*link2* = *p* (tw1)
**if** *p*↑.*link2* ≠ **nil then** *p*↑.*link2*↑.*link1* = *p* (tw2)

We define an implementation scheme, *TwoWayNode*, to express this notion. Structurally, such a node is identical to a *DoubleListNode*. The latter is extended in *TwoWayNode* with attributes *Unlink*, *InsertBefore*, and *Cons* (to override the one in *ListLink*), as well as some protection.

```
scheme TwoWayNode(type ItemType) = dln: DoubleListNode(ItemType)
extended by
    private link1, link2; {Make the attributes which were exported by
        DoubleListNodes, private to users of TwoWayNode.}
    procedure Unlink(p: ref);
        begin
        if p = nil then return nil;
        if p↑.link1 ≠ nil then p↑.link1↑.link2 := p↑.link2;
        if p↑.link2 ≠ nil then p↑.link2↑.link1 := p↑.link1;
        p↑.link1 := nil; {to maintain tw1}
        p↑.link2 := nil; {to maintain tw2}
        end;
    function InsertBefore(p, place: ref) returns ref;
        begin
        if p = nil then return place;
        Unlink(p);
        if place = nil then return p;
        p↑.link2 := place;
        if place↑.link1 ≠ nil then
            begin place↑.link1↑.link2 := p; p↑.link1 := place↑.link1 end;
        place↑.link1 := p;
        return p
        end;
```

```
function Cons(value: ItemType; y: ref) returns ref;
    begin
    var p: ref;
    p := new TwoWayNode;
    p↑.dln.item := value;
    return InsertBefore(p, y);
    end;
end TwoWayNode
```

*Unlink* removes a node from a list, closing up the list around the unlinked node and setting its links to **nil**. *InsertBefore* inserts a node in front of a *place* in a list, modifying the list. *Cons* overrides the procedure of the same name established in *ListLink*. As defined in *TwoWayNode*, *Cons*(*newitem*, *y*) creates a new node to hold *value*, inserts the new node in front of *y*, linking it in so that rules (tw1) and (tw2) hold. With this definition of *Cons*, the other *ListLink* procedures (e.g., *Append* and *MapItem*) work properly and construct well-formed *TwoWayNodes*. *To obtain this effect, we require that when a scheme attribute is overridden, all uses of that name obtain the new definition.*

The specification **private** makes the attributes *link1* and *link2* inaccessible outside the scheme. We claim that this insures that all *TwoWayNodes* are well-formed according to rules (tw1) and (tw2), and reason as follows: The scheme procedures are written so that their outputs satisfy rules (tw1) and (tw2) if their inputs do; the **private** specification insures that the only modifications to the link fields are via scheme procedures; thus, it follows that all *TwoWayNodes* obey rules (tw1) and (tw2) [Wegbreit-Spitzen].

### *4.5 Lists of Reference-counted Objects*

Having defined the schemes for reference counting and lists, it is natural to ask whether the two can be combined to define schemes for lists in which the nodes are reference-counted. We answer in the affirmative and illustrate the case of reference-counted nodes having a single link field.

```
scheme RC(type ItemType) = record
    private type LinkedPart = item: ItemType extended by
        var link: ref;
        type ref = ListLink(ItemType, Node.ref.lower);
        end LinkedPart;
    type Node = ReferenceCounted(LinkedPart);
    type ref = LinkedPart.ref
    end RC;
```

The *RC* scheme defines three type attributes: *LinkedPart*, *Node*, and **ref**. *LinkedPart* is a record type consisting of two components, an *ItemType* and a **ref**. *Node* is defined by the *ReferenceCounted* scheme to be a record consisting of an **integer** *count* and a *LinkedPart*. **ref** is defined as *LinkedPart*.**ref**, which results from instantiating *ListLink*. *RC*(. . .).**ref** is the type to use for declaring pointers to reference-counted objects; e.g.,

```
type RCListOfIntegers = RC(integer).ref;
rc1, rc2: RCListOfIntegers;
```

which defines *rc1* and *rc2* to be pointers to reference counted lists of integers.

### *4.6 Operations on Multiple Scheme Instances*

In the examples so far, a scheme serves to define a set of *scheme instances.* Each instance may have procedure attributes, acting on objects of that instance type. Procedures which span instance boundaries have not yet been considered. For example, operations on lists of integers are complely disjoint from operations on lists of strings. There are cases in which it is convenient to view the instances of a scheme as being more closely related. We now consider schemes *C* having procedure attributes *F* such that *F(x,y)* is well-defined if *x*'s and *y*'s types are instances of *C*, possibly with different scheme parameters. Further, F *has privileged access to both* x *and* y. In this way, a scheme serves as a higher order encapsulation mechanism.

The following scheme for *Complex* illustrates how this is done. *Complex* takes the *FieldType* type as a scheme parameter. This would typically be **real**, but might also be **integer**, *Rational*, **double precision**, or *Algebraic* (for symbolic computations). We give definitions for the structure and a few of the procedure attributes:

```
scheme Complex(type FieldType) = record
    var Re, Im: FieldType

    function Negate(x: Complex) returns z: Complex;
        begin z.Re := Negate(x.Re); z.Im := Negate(x.Im) end;

    formal type t1, t2;

    function PLUS(x: Complex(t1); y: Complex(t2))
                returns z: Complex(PLUS(t1, t2));
        begin
        z.Re := PLUS(x.Re, y.Re); z.Im := PLUS(x.Im, y.Im);
        return z;
        end;
    . . .
    end Complex
```

No INIT procedure is written, so the system will (by default) invoke the INIT for *FieldType* twice - once for the embedded *Re* component and once for *Im.* The code for *Negate* is typical of the procedures used in schemes thus far: the formal parameter *x* has type *Complex.* Each instance of the scheme *Complex* has its own *Negate* procedure in which the token *Complex* in the procedure heading is bound to that instance.

The procedure PLUS is defined to take arguments whose types are any two (possibly different) instances of *Complex.* The variables *t1* and *t2*, which are declared as formals of type **type**, serve as pattern variables in the declaration of PLUS. Such a declaration limits *t1* and *t2* to be **type** values in a pattern match. The formal type expression *Complex(t1)* will match any *Complex* instance and, as part of the pattern matching, the formal type variable *t1* will be assigned a value. This definition of PLUS does not belong to any of the instances; rather, it is distributed across all instance pairs. More properly, it may be thought of as belonging to the scheme itself. Expressing the type returned by *Complex* PLUS also requires new notation which may be read as: Consider the version of PLUS which acts on values of type *t1* and *t2*,. Let *T* be the type it returns. Then PLUS returns a value of type *Complex(T).*

We turn now to a second example of operations on multiple scheme instances. Consider the definition of lists as provided by *ListLink* in section 4.3. As written there, *MapItem* takes a list of *ItemTypes* as its argument and returns a list of *ItemTypes.* This, however, excludes taking a list of integers, applying a function that converts integers into strings, and returning a list of character string representations for the numbers. The

function in such a use maps from integers to strings and is therefore not a legal second argument to *MapItem*. We are going to redefine *MapItem* to take three arguments:

(1) the input
(2) a function which maps from *ItemType* to *ResultType*, and
(3) the type of the result list, *RLinkType*.

The scheme for *ListLink* may be defined or extended to include this definition along with some supporting attributes:

```
formal type ResultType, t3                                          (q1)
function MapItem                                                    (q2)
    (
    x: ListLink;                                                    (q3)
    function F(ItemType) returns ResultType;
    type RLinkType needs RLinkType = ListLink(ResultType, t3)       (q4)
    ) returns RLinkType                                             (q5)
        begin
            if x = nil then return nil                              (q6)
            else return Cons(F(x.item), MapItem(x.link, F, RLinkType));  (q7)
        end;
```

The formal parameters to *MapItem* establish the admissible arguments for its use: *x* must be an instance of *ListLink*. *RLinkType must be the result of instantiating ListLink* with *ResultType* and some acceptable type *t3*. In fact, *t3* only serves as a place holder in the actual parameter list in that mention of *ListLink*.

The constraint on *RLinkType* extends the previous definition of needs: here, the symbol "**needs**" is followed by a Boolean expression comparing two types. The expression must be evaluable at the time a scheme is to be instantiated, and it may involve parameters to that scheme. The earlier expression "**needs attributes**(. . .)" is a special case of this, which can only follow a **type** specification: Boolean combinations of this and other compile-time expressions are also allowed.

The role of *RLinkType* in selecting the appropriate *Cons* routine deserves explication. Together with the types of *x.link* and *F*, it determines the *MapItem* routine to be selected in line (q7); the selected *MapItem*'s result type and *F*'s result type determine the *Cons* routine to be selected.

It is interesting that most of the body of this *MapItem* routine is the same as for the earlier one. Also, observe that both routines may be defined in the same scheme, for they take different arguments, so the generic selection mechanism will choose between them.

A brief discussion of two other uses may help to demonstrate the utility of operations on multiple scheme instances.

(1) It is sometimes useful in Lisp to apply a function of two arguments to corresponding elements of two equal-length lists. The function that does this is called MAP2. In a language with a variety of data types and, in particular, a variety of list types, it is natural to do the same. However, the two lists need not be of the same type. In the scheme system, MAP2 may be defined analogously to the generalized *MapItem*.

(2) Consider defining a scheme for vectors which are to behave like APL arrays of rank 1. The *ItemType* is a formal parameter to the scheme. Unary operations may

be defined in the obvious way. However, binary operations between vectors of *ItemType*-1 and vectors of *ItemType*-2 require the more general mechanism presented above.

## 5. Abstractions and schemes in programming

Abstractions in programming are generally acknowledged to be desirable. To what extent do schemes facilitate the use of abstractions? Viewed as a template for a set of data types, a scheme is an "abstraction" of the types in that set, for it is their generator. Stated somewhat differently, a scheme defines a notion such as "reference-counted nodes of" or "stack of" over all actual parameter types and in this sense it may be regarded as an "abstract" data type.

A scheme may also be viewed as a parameterized implementation technique. For example, a scheme for *BinarySearchTrees* takes the key and value types as parameters; it implements a binary tree in which the keys are maintained in prefix tree order by defining the auxiliary links and the operations on the links and nodes. It does so in a manner suitable for instantiation with any (key-type, value-type) pair, and is thus an "abstract" *BinarySearchTree.*

The above use of the term "abstract" should be distinguished from the idea of "abstraction" as employed in hierarchial program design [Dijkstra, Parnas]. When defining and using an abstraction [Dijkstra2] implemented by an information-hiding module [Parnas], one seeks different ends than those which motivate the use of schemes. As schemes and abstractions are tangentially related, an understanding of their dissimilarities and their exact relation may illuminate certain facets of the scheme system.

To review "abstractions", we consider the paradigm of structured programming [Dijkstra2]. When writing a program at one level, one may use a set of lower level program modules which are treated as "virtual machines". Each virtual machine provides a set of externally available operations (and possibly objects). These machines are generally called "abstractions". An abstraction is characterized completely by its external behavior: how it realizes this behavior is immaterial to users of that abstraction. Indeed, knowledge of the means chosen for its realization may be positively harmful [Parnas]. Several notations, e.g., [Guttag et al.] [Wegbreit-Spitzen, Zilles] have been employed for specifying module behavior in terms that are independent of its internal operation.

The purpose of such abstractions is to decouple a module's specification from its implementation. Since the only thing known about a module at a point of use is the abstract specification, the module implementor is free to choose from a variety of data and control structures when programming it. Moreover, he retains this freedom when rewriting the module at some later time as long as the external interfaces are not changed. Suppose a symbol table module is specified abstractly by stating the types of the keys and values and by expressing the relation between the set of symbol table operations - *InsertKeyValuePair, SearchForKey, DeleteKey,* etc. Then the implementor should be free to realize the symbol table abstraction by an association list, hash table, binary search tree, AVL-tree, or other techniques. Conversely, a user of a symbol table module is free to switch from one realization to another to better the performance of his own program, and he should be able to do this by a simple change of declaration, not by massive program alterations.

It will be seen that abstractions and schemes are independent concepts serving different ends. *A* virtual machine *abstracts a module's specification from its possible implementations. A* scheme *abstracts an implementation technique from the data types which it implements.* What then is the relationship between schemes and abstractions? The concepts touch at two points.

One point of contact is that an abstraction might be implemented by a type produced by a particular scheme instantiation. For example, a symbol table abstraction might be

implemented by a particular instantiation of an AVL-tree scheme. This is *not* to say that schemes are realizations of abstractions. Rather, an instance of a scheme *may be* a realization of an abstraction. This is not hierarchical: A scheme's instances may implement several different abstractions while an abstraction may be implemented by several schemes.

The other point of contact lies farther afield: an abstraction might be used to good purpose to describe the admissible actual parameters corresponding to a scheme formal parameter. This construction would be desirable if a scheme was well-defined only for actual parameter types satisfying certain constraints. For example, consider defining a scheme to implement a paging strategy for a parameterized type *t*. Most of the scheme definition would be concerned with storage allocation, and interface to the file system. It would simplify the definition if it could ignore the issue of how the table which translates from virtual to physical addresses is implemented. One way of deferring this is to write the scheme so that is takes the *TableType* as a parameter. In such a case, it would be desirable to state in the scheme definition that *TableType* can be bound only to those types which satisfy a given abstract specification of how a *TableType* must behave. This is not to suggest that the compiler attempt to prove that a given type realizes an abstraction - such questions are undecidable in general and, with today's theorem proving technology, are intractible for any but the simplest of cases. Rather we point out the desirability of providing constructs in a programming language to

(1) specify that an actual parameter must satisfy certain abstract specifications, and

(2) assert that a given type does satisfy certain abstract specifications.

With such features, abstractions serve as part of the pattern-matching language for scheme parameter specifications.

## 6. Specifying constraints on parameters

In the discussion of linked lists in sections 4.3 and 4.6, we used the notion of a *needs specification*. It was used to limit the range of acceptable types which could be used as parameters to a scheme to those having at least the attributes given in the specification. It is often useful to employ a needs specification in a somewhat different way: to choose from a set of entities (whether procedures or schemes) the one to be selected in a particular context.

The idea is simple: extend generic selection to schemes. Several schemes may have the same name, say *S*. (This is particularly useful when the several schemes provide alternative implementation techniques for some common abstraction.) When the name appears in an instantiation, e.g.,

*t:* **type** = $S(\alpha)$,

the generic mechanism selects the appropriate scheme based on the actual parameters $\alpha$: that **scheme** is chosen whose **needs** specification is satisfied by $\alpha$. The **needs** specification serves to discriminate among generic schemes just as type structure discriminates among generic procedures: It may be any Boolean expression which can be evaluated at compile time; the form **needs attributes** (...) is a special case of this.

Consider the symbol table notion mentioned above. If one is to choose among a number of alternative implementations depending on, say, performance desiderata, then there must be some form of documentation to help the user choose. For instance, a hash table might be the preferred choice, depending on the key or value types, and on the number of (key, value) pairs to be stored in the table. One might discriminate more finely and pick one of a number of different hash table disciplines depending on the number of items,

space constraints, etc. The following three schemes all define symbol tables, and have the same name; by using needs specifications, we make them all take the same number and type of parameters, with the best one for a given scheme instantiation being chosen generically:

```
scheme SymbolTable                                                    (s1)
        (
        type keytype, valuetype;
        ordered: Boolean;
        maxEntries: integer needs (maxEntries in 11..1000)
        ) =
    HashTable(keytype, valuetype, maxEntries)
    end SymbolTable;

scheme SymbolTable                                                    (s2)
        (
        type keytype, valuetype;
        ordered: Boolean needs ordered;
        maxEntries: integer needs (maxEntries > 1000)
        ) =
    AVLTree(keytype, valuetype)
    end SymbolTable;

scheme SymbolTable                                                    (s3)
        (
        type keytype, valuetype;
        ordered: Boolean;
        maxEntries: integer needs (maxEntries in 0..10)
        ) =
    LinearSymbolTable(keytype, valuetype, maxEntries)
    end SymbolTable;
```

Scheme definition (s1) basically says: if you need a symbol table package in which *ordering* is not constrained and for which the maximum number of entries is in the range 11..1000, then use some *HashTable* scheme selected by the same parameters (we will give a number of scheme definitions for *HashTable* below). Scheme (s2) says: if the symbol table must be ordered, and there are more than 1000 items to be stored, then an AVL-tree implementation is the one of choice. Lastly, scheme definition (s3) says: just use a simple *LinearSymbolTable* if there are only a few (less than 11), items to be stored.

Simple constraints on the range of an integer variable have previously been used in the type machinery of a number of languages such as Pascal [Wirth], but there are two difficulties [Habermann]:

(1) types are intended to be checkable at compile time, but assignment of an integer value to a restricted range variable really requires run-time checking or verification [Euclid][Suzuki-Ishihata];

(2) the only way to relate type definitions is by nested definition, which forces a strict hierarchy among defined types, whereas subrange declarations can form a lattice.

We propose instead to treat such constraints separately from the type structure; thus, *maxEntries* in the above declarations is simply an integer which is constrained to a specific range in each declaration. This does not eliminate the need for run-time checks, but does factor it out from type checking. In the above case, the constraints must be satisfied at scheme instantiation time, and we will continue to assume compile-time constraint checking in this section.

The next, skeletal schemes, each named *HashTable*, are intended as alternative definitions suitable for use with the *SymbolTable* scheme (s1), and will further constrain the possible values of type parameters:

```
scheme HashTable                                                    (h1)
        (
        type keytype needs keytype = string or keytype = integer;
        type valuetype;
        maxEntries: integer needs maxEntries in 11..100
        ) =
        {simple linear-probe hash table}
    . . .
    end HashTable;
scheme HashTable                                                    (h2)
        (
        type keytype
            needs attributes(procedure Hash(keytype) returns 0..maxEntries-1);
        type valuetype;
        maxEntries: integer needs maxEntries in 101..500;
        ) =
        {linear probe table keeping value stored directly with key}
    . . .
    end HashTable;
scheme HashTable                                                    (h3)
        type keytype
            needs attributes(procedure Hash(keytype) returns 0..maxEntries-1);
        type valuetype;
        maxEntries: integer needs maxEntries in 501..1000
        ) =
        {chained hash table to reduce collisions at the expense of space}
    . . .
    end HashTable;
```

Scheme definition (h1) constrains the *keytype* parameter to be either **string** or **integer** by using **needs** and the equality relation on types (we assume that the scheme provides a hashing function for these types). Definition (h2) uses the "**attributes**" operator in a needs specification to indicate that this *HashTable* scheme requires a user-supplied *Hash* function. Lastly, (h3) is intended to be used for large tables.

## 7. Implementation notes

In this section we lay out what appear to be the main issues in implementing schemes and sketch ways in which these issues may be handled. One of the necessary properties of the scheme system is that schemes are fully resolved during compilation: They determine what code is produced, but schemes *per se* appear nowhere in the runtime environment. Thus our discussion is confined to how the compiler deals with schemes. In order to confine this discussion to the prominent points, we make the simplifying assumption that all procedures are compiled together. We ignore issues which arise in loading and binding separately compiled modules.

Four language constructs require special treatment because of schemes:

(1) scheme definitions, e.g., **scheme** *Queue*(**type** *ItemType*) = . . .

(2) scheme instantiations, e.g., **type** *IntQueue* = *Queue*(**integer**);

(3) procedure calls in which the procedure name is a scheme attribute name, e.g., *Add* in *Add*(3, *q*)

(4) procedure calls in which an argument is a scheme attribute name, e.g., *Add* in *Foo*(*q*, *x*, *Add*)

We consider these constructs in turn.

### *7.1 Scheme Definitions*

A scheme definition is processed to build tables which facilitate instantiation, and to check so far as possible that the scheme is type-correct; however, no code is generated. Checking type-correctness of a scheme is desirable in order to detect errors early - at the point of definition - rather than later in an instantiation where the fault may be confounded with errors of instantiation. The **needs** specification on the scheme formal parameters provides information which allows partial checking of the scheme's body. However, it is not always possible to make a complete type check on the basis of the definition alone.

For example, the scheme for *ListLink* in section 4.3 is written with a **needs** clause which specifies that the argument *PtrType* has attributes *link* and *item* with certain type structures. The specification for *PtrType* is so written that the type correctness of expressions such as *p.item* in *Cons* can be checked from its definition alone; any admissible *PtrType* must result in a *ListLink* instance having a type-correct version of *Cons*. However, similar considerations cannot be applied to the definition of PLUS in the scheme *Complex* (section 4.6). We want PLUS to be defined *whenever used* for all pairs of *Complex* instances. It would be too restrictive to demand in a **needs** clause that there be a PLUS routine for each pair of *Field* types used as arguments. Many of these might never be used in the actual program.

The issue is a conflict between two features, both of which are desirable:

(1) specifying precisely the requirements a scheme imposes on its arguments, and

(2) imposing no more constraints than actually required by the use of a scheme instance.

We intend that the **needs** specification be used to strike the appropriate balance in each scheme definition. The writer of the scheme specifies criteria for admissible parameters. The compiler will type-check the scheme using the information they provide. Any expressions in a scheme which are demonstrably type-incorrect are flagged as errors. Those expressions which can be certified as type-correct are all to the good. The others will have to be checked for type-correctness in each instantiation. In any case, compile-type type checking is complete; the only issue is where it takes place.

### *7.2 Scheme Instantiations*

A scheme instantiation results in a type definition which can be treated much like any other type definition. Processing an instantiation can be loosely thought of as first reducing the scheme instantiation to a type, and then processing the type. Actually, the processing is complicated by some additional semantic issues and a few space-optimization considerations.

When a scheme instantiation is encountered, actual parameters are substituted for formal parameters in the scheme definition, and constraints on the parameter specifications

are checked. If the actual parameters form an admissible binding set, the scheme body is instantiated with these bindings, and the body is checked for type correctness.

As a matter of implementation strategy, it seems desirable not to treat each instance as a completely separate type nor to generate code for each scheme procedure in an instance. There are two reasons for this:

(1) A scheme may export a large number of procedure attributes, but many of these may never be used in a particular program for a particular instance. We would like to compile code and allocate space only for those procedure instances that are actually needed.

(2) It may be possible to use the same piece of code to implement a given procedure in several different scheme instances if these instances have the same physical layout. In this case, the several scheme instances form an *equivalence class* with respect to the implementation of the given procedure. We want all members of the equivalence class to share one code block.

We suggest that a scheme instance be processed only to check the admissibility of the instantiating arguments and to build tables, e.g., those describing physical storage layout, attribute names and their types, etc. The entire program is processed to determine which procedure instances are actually needed and which instances fall into the same equivalence class. On a subsequent pass, code can be generated as actually needed and shared where possible.

### *7.3 Procedure Calls in which the Operator is a Scheme Attribute Name*

Whenever a procedure call occurs, the generic selection mechanism uses type structure to choose a single meaning for the procedure name from the possible set of procedures, some of which may be scheme attributes. Scheme procedures may be divided into two groups, depending on their parameter specifications: those which are fully instantiated in a scheme instance and those containing unbound formal types (section 4.6) in their parameter specifications..

Fully instantiated scheme procedures are the simpler case. For example, the procedure *Negate* in the scheme *Complex* has the type specification

**procedure** *Negate*(*Complex*) **returns** *Complex*

An instantiation of *Complex,* e.g.,

*RC*: *Complex*(**integer**)

behaves *as if* an instance of *Negate* were created such that *Complex* in the specification is bound to the type *Complex*(**integer**). Choosing this instance upon encountering a procedure call, e.g.,

*Negate*(*c*)

where *c* has type *RC*, entails normal application of the generic selection mechanism. Recall that code generation is deferred, so a table describing instance types is maintained separately from any code.

Scheme procedures that contain unbound formal types require more involved processing. For example, the procedure PLUS in the scheme *Complex* has the type specification

**procedure** PLUS(*Complex*(*t1*), *Complex*(*t2*)) **returns** *Complex*(PLUS(*t1*, *t2*))

where *t1* and *t2* are unbound scheme formals. Suppose *c1* and *c2* have type *RC* as defined above. We consider the processing of

PLUS(*c1, c2*).

The relevant symbol table information may be summarized as

*c1* :: type is *RC*;

*c2* :: type is *RC*;

*RC* :: type is **type**; value is <*some type table entry*>; is an instance of *Complex* with *FieldType* bound to **integer**

PLUS :: generic procedure name; possible interpretations are:

. . . *Complex*.PLUS . . .

The generic selection mechanism examines the list of possible interpretations for PLUS, looking for those having a type structure compatible with the argument type structure, here (*RC, RC*) which is equal to (*Complex*(*integer*), *Complex*(*integer*)). Since *RC* is an instance of *Complex* with *FieldType* bound to **integer** and since the formal parameter list for *Complex*.PLUS has the type structure (*Complex*(*t1*), *Complex*(*t2*)), this is a match with *t1* and *t2* each bound to **integer.** If there is a single admissible interpretation for PLUS, then it is used. If there were no admissible interpretations, then the program would be incomplete. If there were more than one, the program would be ambiguous.

When a scheme procedure is selected, two activities are performed:

(1) The compiler checks to see if a suitable code body has already been generated (either because the procedure has been generated for this scheme instance or because a shared code body has been generated). If not, code is generated and becomes available to all procedures in the implementation equivalence class.

(2) The **returns** specification for the selected procedure determines the type of the result. In the case of scheme procedures with unbound formals, this may require binding any formals which occur in the **returns** specification and invoking the generic selection machinery. For example, to process the specification **returns** *Complex*(PLUS(*t1, t2*)), the bindings for *t1* and *t2* are obtained; call these *t1** and *t2**. The generic selection mechanism is invoked to find meanings for PLUS with argument structure (*t1*, t2**). Suppose a unique meaning is found and that it specifies **returns** *t3**. Then, **returns** *Complex*(PLUS(*t1, t2*)) reduces to **returns** *Complex*(*t3**).

### *7.4 Procedure Calls Where an Argument is a Scheme Procedure Name*

The situation is more involved when the meaning of an argument can be determined only by considering the calling procedure. This occurs when a generic procedure is used as a procedure-valued argument in a procedure call. For example,

*Foo*(13, PLUS)

Suppose *Foo* has the type specification

**procedure** *Foo*(**integer**;
**procedure**(*Complex*(**real**), *Complex*(**real**)) **returns** *Complex*(**real**));

The type of *Foo*'s second formal parameter dictates that *Complex*(**real**).PLUS be chosen as the meaning of the second argument to *Foo.*

The issue is further complicated if the calling procedure name is itself generic. In the above example, suppose that *Foo* has two interpretations - the one shown and another with the type structure

```
procedure Foo(integer;
              procedure(string) returns string);
```

We adopt the following rule: All possible meanings for caller and argument are considered and their type specifications checked for type consistency. If exactly one interpretation is type-consistent, then that one is chosen. Otherwise, the expression is considered to be in error.

Observe that these program phenomena and our need for selection rules result from generic procedure names being used as arguments to procedures. Whether the procedures are scheme attributes or not does not effect the selection rule; it does increase the number of cases to be considered in its implementation.

## 8. Summary

We have developed a notation and semantics for parametrized implementation mechanisms called *schemes*. Schemes are built upon five basic notions:

- extensible data types,
- the ability to refer to the attributes of an object in a uniform way,
- the ability to override the system-defined meanings of attributes,
- the ability to encapsulate a set of definitions, and
- the ability to control external access to the attributes of an encapsulated set.

Schemes extend the state of the art for generalizing implementation techniques to serve a wide variety of applications. Once a scheme such as *ListLink* is correctly programmed, that concept need not be coded again. This may help to make safe and palatable implementation techniques which involve low-level, but powerful programming concepts such as pointers. The user of a scheme may be assured of its correctness; and encapsulation may be used to ensure that it cannot be misused.

Viewed from outside, a scheme can be regarded as providing one possible implementation of some abstract specification. A set of schemes may be used to provide alternative realizations of a single abstraction having different performance characteristics. It proves convenient to give all the schemes in the set a common name, and to use constraint specifications to choose among them based on the properties of a particular use.

Type errors can occur in schemes, their instantiations, and the uses of these instances. Since it is desirable to detect and report errors as early as possible, the **needs** specification was introduced as a means for doing limited type checking within schemes, and for checking the parameters to a scheme instance. It was shown that complete error checking within the scheme imposes an overly strong constraint on the programmer; hence, some of the error checking occurs when scheme instances are used. All type checking is, however, carried out during compilation.

Finally, we have discussed compilation techniques for schemes. We outlined delayed code generation and considerations of code sharing among separate scheme instances.

## 9. Acknowledgements

This work has benefitted from reviews by a number of our colleagues, including Butler Lampson and Jim Morris, and the members of the ARPA Quality Software group. We are also indebted to Gail Pilkington for the editing and preparation of this report.

## REFERENCES

[Dahl]
Dahl, O. J., Myhrhaug, B. and Nygaard. K. *The Simula 67 Common Base Language*, Publication S-22, Norwegian Computing Center, Oslo, 1970.

[Dijkstra]
Dijkstra, E. W. The Structure of the "THE"-Multiprogramming System, *Comm. ACM, Vol. 11, No. 5* (May 1968) pp. 341-346.

[Dijkstra2]
Dijkstra, E. W. Notes on Structured Programming, in *Structured Programming, A.P.I.C. Studies in Data Processing, No. 8*, Academic Press (1972), pp. 1-82.

[Euclid1]
Lampson, B.W., Horning, J.J., London, R.L., Mitchell, J.G., and Popek, G.J. Report on the Programming Language Euclid, *SIGPLAN Notices* (February 1977)

[Geschke-Mitchell]
Geschke, C. M. and Mitchell, J. G. On the Problem of Uniform References to Data Structures, *IEEE Trans. on Software Engineering, Vol. SE-1, No. 2* (June 1975) pp. 207-219.

[Gries-Gehani]
Gries, D. and Gehani, N. Some Ideas on Data Types in High Level Languages, to appear in *Comm. ACM, Vol. 20.*

[Habermann]
Habermann, A. N. Critical Comments on the Programming Language Pascal, *Acta Informatica 3*, (1973), pp. 47-57.

[Hoare]
Hoare, C.A.R. Recursive Data Structures, *Int. Jrnl. of Computer and Information Sciences, Vol. 4, No. 2* (June 1975) pp. 105-132.

[Guttag-Horowitz-Musser]
Guttag, J. V., Horowitz, E. and Musser, D. The Design of Data Type Specifications, USC Technical Report ISI/RR-76-49, November 1976.

[IBM1]
IBM System/360 Operating System, *PL/I Language Specifications*, Form C28-6571-4, New York, 1966.

[Liskov-Zilles]
Liskov, B. and Zilles, S. Programming with Abstract Data Types, *Proc. of ACM SIGPLAN Conf. on Very High Level Languages, SIGPLAN Notices, Vol. 9, No. 4* (April 1974) pp. 50-59.

[McCarthy]
McCarthy, J. Recursive Functions of Symbolic Expressions and their Computation by Machine, *Comm. ACM, Vol. 3, No. 4* (April 1960) pp. 184-195.

[Morris]

Morris, J. H. Types Are Not Sets, *SIGPLAN Symposium on Principles of Programming Languages* (Oct. 1973) pp. 1-5.

[Palme]
Palme, J. Protected Program Modules in Simula 67, Research Institute of National Defense, Stockholm, Sweden (July 1973).

[Parnas]
Parnas, D. L. Information Distribution Aspects of Design Methodology, *Proc. IFIP Congress 1971*, pp. 339-344.

[Richards]
Richards, M. BCPL: A Tool for Compiler and System Writing, *AFIPS Conf. Proc. 1969 SJCC, Vol. 34*, pp. 557-566.

[Suzuki-Ishihata]
Suzuki, N. and Ishihata, K. Implementation of Array Bound Checker, *SIGPLAN Symposium on Principles of Programming Languages* (Jan 1977) pp. 132-143.

[vanWijngaarden]
van Wijngaarden A. Report on the Algorithmic Language Algol 68, *Numerische Mathematik, Vol. 14* (1969) pp. 79-218.

[Wegbreit]
Wegbreit, B. The Treatment of Data Types in EL1, *Comm. of the ACM, Vol. 17, No. 5* (May 1974) pp. 251-264.

[Wegbreit-Spitzen]
Wegbreit, B. and Spitzen, J. Proving Properties of Complex Data Structures, to appear in the *Journal of the ACM.*

[Wirth]
Wirth, N. An Assessment of the Programming Language Pascal, Proc. *Inter. Conf. on Reliable Software*, April 1975, pp. 23-30.

[Wulf]
Wulf, W. ALPHARD: Toward a Language to Support Structured Programs, Computer Science Dept., Carnegie-Mellon University, Pittsburgh, Pa. April, 1974.

[Wulf2]
Wulf, W. *et. al. The Design of an Optimizing Compiler*, American Elsevier Publishing Co., Inc., New York, 1975.

[Zilles]
Zilles, S. Algebraic Specifications for Data Types, IBM Research Laboratory, San Jose, Calif. January, 1975.

XEROX

XEROX

Schemes: A High Level Data Structuring Concept by James G. Mitchell and Ben Wegbreit

Xerox Corporation
Palo Alto Research Center
3333 Coyote Hill Road
Palo Alto, California 94304

XEROX® is a trademark of XEROX CORPORATION Printed in U.S.A.

CSL-77-1